A Naive Prover for First-Order Logic

Asta Halkjær From

Technical University of Denmark, 2022-04-29

Syntax

Terms: variables, functions. *The same de Bruijn indices as in SeCaV*

```
datatype tm
  = Var nat (‹#›)
  | Fun nat ‹tm list› (‹†›)
```

Formulas: falsity, predicates, implication, universal quantification

```
datatype fm
  = Falsity (‹⊥›)
  | Pre nat ‹tm list› (‹‡›)
  | Imp fm fm (infixr ‹⟶› 55)
  | Uni fm (‹∀›)

type_synonym sequent = ‹fm list × fm list›
```

Sequent Calculus

$$\textsc{Idle}\ \frac{A \vdash B}{A \vdash B} \qquad \textsc{Axiom}\ n\ ts\ \frac{}{A \vdash B}\ \textsc{if}\ \ddagger n\ ts\ [\in]\ A\ \textsc{and}\ \ddagger n\ ts\ [\in]\ B$$

$$\textsc{FlsL}\ \frac{}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ A \qquad \textsc{FlsR}\ \frac{A \vdash B\ [\div]\ \bot}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ B$$

$$\textsc{ImpL}\ p\ q\ \frac{A\ [\div]\ (p \longrightarrow q) \vdash p\ \#\ B \qquad q\ \#\ A\ [\div]\ (p \longrightarrow q) \vdash B}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ A$$

$$\textsc{ImpR}\ p\ q\ \frac{p\ \#\ A \vdash q\ \#\ B\ [\div]\ (p \longrightarrow q)}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ B$$

$$\textsc{UniL}\ t\ p\ \frac{p\langle t/0\rangle\ \#\ A \vdash B}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ A \qquad \textsc{UniR}\ p\ \frac{A \vdash p\langle \#\mathit{fresh}(A@B)/0\rangle\ \#\ B\ [\div]\ \forall p}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ B$$

Prover Output

```
|- ((P) --> (Falsity)) --> ((P) --> (Falsity))
 + ImpR on (P) --> (Falsity) and (P) --> (Falsity)
(P) --> (Falsity) |- (P) --> (Falsity)
 + ImpR on P and Falsity
P, (P) --> (Falsity) |- Falsity
 + ImpL on P and Falsity
  P |- P, Falsity
   + FlsR
  P |- P
   + Axiom on P
  Falsity, P |- Falsity
   + FlsL
```

Prover Idea I

A *stream of rules* tells us what to do

- Say we have the sequent $\vdash$ †0 [] $\longrightarrow$ †0 []
- The rule ImpR (†0 []) (†0 []) says we can prove it *if*
- we can prove the sequent †0 [] $\vdash$ †0 []

Thus we need to ensure that we always *eventually* reach the rule we need

- We need to reach Axiom 0 [] for the sequent †0 [] $\vdash$ †0 []
- But Axiom 1 [] doesn’t harm us

Prover Idea II

Consider the stream of numbers (pretend they are rules):

0 1 2 3 4 5 6 7 8 9 10 11 12 …

Every number appears somewhere in the sequence

So we will reach the number we need at some point!

But what if we need it twice? Or we need 12 before we need 5?

Prover Idea III

Consider the stream of numbers

0 0 1 0 1 2 0 1 2 3 0 1 2 3 4 ...

No matter how many times a number has already appeared, it keeps appearing

The stream is *fair* (but larger numbers are further away than before)

How to get a fair stream of *rules*?

My Theory Fair-Stream

```
definition upt_lists :: ‹nat list stream› where
  ‹upt_lists ≡ smap (upt 0) (stl nats)›
```

[0] [0, 1] [0, 1, 2] [0, 1, 2, 3] …

```
definition fair_nats :: ‹nat stream› where
  ‹fair_nats ≡ flat upt_lists›
```

0, 0, 1, 0, 1, 2, 0, 1, 2, 3, …

```
definition fair :: ‹'a stream ⇒ bool› where
  ‹fair s ≡ ∀x ∈ sset s. ∀m. ∃n ≥ m. s !! n = x›
```

A handful of lemmas later…

```
definition fair_stream :: ‹(nat ⇒ 'a) ⇒ 'a stream› where
  ‹fair_stream f = smap f fair_nats›

theorem fair_stream: ‹surj f ⟹ fair (fair_stream f)›
  unfolding fair_stream_def using fair_surj .

theorem UNIV_stream: ‹surj f ⟹ sset (fair_stream f) = UNIV›
  unfolding fair_stream_def using all_ex_fair_nats by (metis sset_range stream.set_map surjI)
```

Encoding To and From the Natural Numbers

The Isabelle theory Nat-Bijection provides the following operations:

- prod_encode :: "nat × nat ⇒ nat"
- prod_decode :: "nat ⇒ nat × nat"
- sum_encode :: "nat + nat ⇒ nat"
- sum_decode :: "nat ⇒ nat + nat"
- list_encode :: "nat list ⇒ nat"
- list_decode :: "nat ⇒ nat list"

I write ‹c $ x ≡ sum_encode (c x)›

Encoding Terms as Natural Numbers

```
primrec nat_of_tm :: ‹tm ⇒ nat› where
  ‹nat_of_tm (#n) = prod_encode (n, 0)›
| ‹nat_of_tm (†f ts) = prod_encode (f, Suc (list_encode (map nat_of_tm ts)))›

function tm_of_nat :: ‹nat ⇒ tm› where
  ‹tm_of_nat n = (case prod_decode n of
    (n, 0) ⇒ #n
  | (f, Suc ts) ⇒ †f (map tm_of_nat (list_decode ts)))›
  by pat_completeness auto
termination by (relation ‹measure id›) simp_all

lemma tm_nat: ‹tm_of_nat (nat_of_tm t) = t›
  by (induct t) (simp_all add: map_idI)

lemma surj_tm_of_nat: ‹surj tm_of_nat›
  unfolding surj_def using tm_nat by metis
```

Encoding Formulas as Natural Numbers

```
primrec nat_of_fm :: ‹fm ⇒ nat› where
  ‹nat_of_fm ⊥ = 0›
| ‹nat_of_fm (‡P ts) = Suc (Inl $ prod_encode (P, list_encode (map nat_of_tm ts)))›
| ‹nat_of_fm (p ⟶ q) = Suc (Inr $ prod_encode (Suc (nat_of_fm p), nat_of_fm q))›
| ‹nat_of_fm (∀p) = Suc (Inr $ prod_encode (0, nat_of_fm p))›

function fm_of_nat :: ‹nat ⇒ fm› where
  ‹fm_of_nat 0 = ⊥›
| ‹fm_of_nat (Suc n) = (case sum_decode n of
    Inl n ⇒ let (P, ts) = prod_decode n in ‡P (map tm_of_nat (list_decode ts))
  | Inr n ⇒ (case prod_decode n of
      (Suc p, q) ⇒ fm_of_nat p ⟶ fm_of_nat q
    | (0, p) ⇒ ∀(fm_of_nat p)))›
  by pat_completeness auto
termination by (relation ‹measure id›) simp_all

lemma fm_nat: ‹fm_of_nat (nat_of_fm p) = p›
  using tm_nat by (induct p) (simp_all add: map_idI)

lemma surj_fm_of_nat: ‹surj fm_of_nat›
  unfolding surj_def using fm_nat by metis
```

Encoding Rules as Natural Numbers

```
definition idle_nat :: nat where
  ‹idle_nat ≡ 4294967295›

primrec nat_of_rule :: ‹rule ⇒ nat› where
  ‹nat_of_rule Idle = Inl $ prod_encode (0, idle_nat)›
| ‹nat_of_rule (Axiom n ts) = Inl $ prod_encode (Suc n, Suc (list_encode (map nat_of_tm ts)))›
| ‹nat_of_rule FlsL = Inl $ prod_encode (0, 0)›
| ‹nat_of_rule FlsR = Inl $ prod_encode (0, Suc 0)›
| ‹nat_of_rule (ImpL p q) = Inr $ prod_encode (Inl $ nat_of_fm p, Inl $ nat_of_fm q)›
| ‹nat_of_rule (ImpR p q) = Inr $ prod_encode (Inr $ nat_of_fm p, nat_of_fm q)›
| ‹nat_of_rule (UniL t p) = Inr $ prod_encode (Inl $ nat_of_tm t, Inr $ nat_of_fm p)›
| ‹nat_of_rule (UniR p) = Inl $ prod_encode (Suc (nat_of_fm p), 0)›
```

```
lemma ‹map rule_of_nat [0..<100] =
 [FlsL, ImpL ⊥ ⊥, FlsR, UniL (# 0) ⊥, UniR ⊥, ImpR ⊥ ⊥, ImpR (‡ 0 []) ⊥,
  ImpL ⊥ (‡ 0 []), Axiom 0 [], ImpR ⊥ (‡ 0 []), UniR (‡ 0 []),
  ImpL (‡ 0 []) ⊥, ImpR ⊥ (∀ ⊥), UniL (# 0) (‡ 0 []), Axiom 0 [# 0],
  ImpR ⊥ (∀ ⊥), Axiom 1 [], UniL († 0 []) ⊥, UniR (∀ ⊥), ImpR (‡ 0 []) ⊥,
  ImpR (‡ 0 []) (‡ 0 []), ImpL ⊥ (∀ ⊥), Axiom 0 [# 0, # 0],
  ImpR ⊥ (‡ 0 [# 0]), Axiom 1 [# 0], ImpL (‡ 0 []) (‡ 0 []), Axiom 2 [],
  ImpR (‡ 0 []) (‡ 0 []), UniR (‡ 0 [# 0]), ImpL (∀ ⊥) ⊥, ImpR (∀ ⊥) ⊥,
  UniL (# 0) (∀ ⊥), Axiom 0 [† 0 []], ImpR ⊥ (∀ (‡ 0 [])),
  Axiom 1 [# 0, # 0], UniL († 0 []) (‡ 0 []), Axiom 2 [# 0],
  ImpR (‡ 0 []) (∀ ⊥), Axiom 3 [], UniL (# 1) ⊥, UniR (∀ (‡ 0 [])),
  ImpR (∀ ⊥) ⊥, ImpR ⊥ (‡ 0 [# 0]), ImpL ⊥ (‡ 0 [# 0]),
  Axiom 0 [# 0, # 0, # 0], ImpR ⊥ (‡ 1 []), Axiom 1 [† 0 []],
  ImpL (‡ 0 []) (∀ ⊥), Axiom 2 [# 0, # 0], ImpR (‡ 0 []) (‡ 0 [# 0]),
  Axiom 3 [# 0], ImpL (∀ ⊥) (‡ 0 []), Axiom 4 [], ImpR (∀ ⊥) (‡ 0 []),
  UniR (‡ 1 []), ImpL (‡ 0 [# 0]) ⊥, ImpR (‡ 0 []) (∀ ⊥),
  UniL (# 0) (‡ 0 [# 0]), Axiom 0 [† 0 [], # 0], ImpR ⊥ (⊥ ⟶ ⊥),
  Axiom 1 [# 0, # 0, # 0], UniL († 0 []) (∀ ⊥), Axiom 2 [† 0 []],
  ImpR (‡ 0 []) (∀ (‡ 0 [])), Axiom 3 [# 0, # 0], UniL (# 1) (‡ 0 []),
  Axiom 4 [# 0], ImpR (∀ ⊥) (∀ ⊥), Axiom 5 [], UniL († 0 [# 0]) ⊥,
  UniR (⊥ ⟶ ⊥), ImpR (‡ 0 [# 0]) ⊥, ImpR (∀ ⊥) (‡ 0 []),
  ImpL ⊥ (∀ (‡ 0 [])), Axiom 0 [# 1], ImpR ⊥ (‡ 0 [# 0, # 0]),
  Axiom 1 [† 0 [], # 0], ImpL (‡ 0 []) (‡ 0 [# 0]), Axiom 2 [# 0, # 0, # 0],
  ImpR (‡ 0 []) (‡ 1 []), Axiom 3 [† 0 []], ImpL (∀ ⊥) (∀ ⊥),
```

What Does It Matter? I

```
term ‹P ⟶ P›
term ‹‡0 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡0 []) = 1› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡0 [])) = 27› by eval
lemma ‹nat_of_rule (Axiom 0 []) = 8› by eval

term ‹(∀x. P x) ⟶ P a›
term ‹∀(‡0 [#0]) ⟶ ‡0 [†0 []]›
lemma ‹nat_of_fm (‡0 [†0 []]) = 13› by eval
lemma ‹nat_of_rule (ImpR (∀(‡0 [#0])) (‡0 [†0 []])) = 1865› by eval
lemma ‹nat_of_rule (UniL (†0 []) (∀(‡0 [#0]))) = 997› by eval
lemma ‹nat_of_rule (Axiom 0 [†0 []]) = 32› by eval
```

Recall what the sequence looks like: 0 0 1 0 1 2 0 1 2 3 …

We reach 1865 only at position 1865*(1+1865)/2 = **1740045**.

What Does It Matter? II

The numbers in the formulas matter:

```
term ‹P ⟶ P ⟶ P›
term ‹‡0 [] ⟶ ‡0 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡0 [] ⟶ ‡0 []) = 18› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡0 [] ⟶ ‡0 [])) = 469› by eval

term ‹P ⟶ Q ⟶ P›
term ‹‡0 [] ⟶ ‡1 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡1 [] ⟶ ‡0 []) = 70› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡1 [] ⟶ ‡0 [])) = 5409› by eval
```

We reach 469 at position **110215**

We reach 5409 at position **14631345**

Example Proofs I

```
time ./Main "Imp (Pre 0 []) (Pre 0 [])"
|- (P) --> (P)
 + ImpR on P and P
P |- P
 + Axiom on P
```

```
Executed in    9.80 millis
```

Example Proofs II

```
time ./Main "Imp (Uni (Pre 0 [Var 0])) (Pre 0 [Fun 0 []])"
|- (forall P(0)) --> (P(a))
 + ImpR on forall P(0) and P(a)
forall P(0) |- P(a)
 + UniL on 0 and P(0)
P(0), forall P(0) |- P(a)
 + UniL on a and P(0)
P(a), P(0), forall P(0) |- P(a)
 + UniL on 1 and P(0)
P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0) and P(0)
P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on b and P(0)
P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 2 and P(0)
P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0, 0) and P(0)
P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(0) and P(0)
P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on c and P(0)
P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 3 and P(0)
P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(a) and P(0)
P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(0, 0) and P(0)
P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on h(0) and P(0)
P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on d and P(0)
P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 4 and P(0)
P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0, 0, 0) and P(0)
P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(a) and P(0)
P(g(a)), P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on h(0, 0) and P(0)
P(h(0, 0)), P(g(a)), P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)),
P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + Axiom on P(a)
________________________________________
Executed in  3.51 secs
```

We need to get to **1865** to hit the ImpR rule.

Then we start back at 0.

The UniL rule we need is at **997**.

But then we keep running from **997** to **1866**.

And hit lots of **UniL** rules in between…

In the end: *a very silly derivation*.

Example Proofs III

```
time ./Main "Imp (Pre 0 []) (Imp (Pre 0 []) (Pre 0 []))"
|- (P) --> ((P) --> (P))
 + ImpR on P and (P) --> (P)
P |- (P) --> (P)
 + ImpR on P and P
P, P |- P
 + Axiom on P
```

(position **110215**)

`Executed in` **`192.72 millis`**

Example Proofs IV

```
time ./Main "Imp (Pre 0 []) (Imp (Pre 1 []) (Pre 0 []))"
|- (P) --> ((Q) --> (P))
 + ImpR on P and (Q) --> (P)
P |- (Q) --> (P)
 + ImpR on Q and P
Q, P |- P
 + Axiom on P
```

(position **14631345**)

Executed in **43.01 secs**

Isabelle/HOL Details I

```
datatype rule
  = Idle
  | Axiom nat ‹tm list›
  | FlsL
  | FlsR
  | ImpL fm fm
  | ImpR fm fm
  | UniL tm fm
  | UniR fm

definition rules :: ‹rule stream› where
  ‹rules ≡ fair_stream rule_of_nat›

lemma UNIV_rules: ‹sset rules = UNIV›
  unfolding rules_def using UNIV_stream surj_rule_of_nat .
```

A datatype for our rules

A fair stream of rules

which includes every rule

Sequent Calculus Reprise

$$\textsc{Idle}\ \frac{A \vdash B}{A \vdash B} \qquad \textsc{Axiom}\ n\ ts\ \frac{}{A \vdash B}\ \textsc{if}\ \ddagger n\ ts\ [\in]\ A\ \textsc{and}\ \ddagger n\ ts\ [\in]\ B$$

$$\textsc{FlsL}\ \frac{}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ A \qquad \textsc{FlsR}\ \frac{A \vdash B\ [\div]\ \bot}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ B$$

$$\textsc{ImpL}\ p\ q\ \frac{A\ [\div]\ (p \longrightarrow q) \vdash p \# B \qquad q \# A\ [\div]\ (p \longrightarrow q) \vdash B}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ A$$

$$\textsc{ImpR}\ p\ q\ \frac{p \# A \vdash q \# B\ [\div]\ (p \longrightarrow q)}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ B$$

$$\textsc{UniL}\ t\ p\ \frac{p\langle t/0\rangle \# A \vdash B}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ A \qquad \textsc{UniR}\ p\ \frac{A \vdash p\langle \# \mathit{fresh}(A@B)/0\rangle \# B\ [\div]\ \forall p}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ B$$

Isabelle/HOL Details II

```
function eff :: ‹rule ⇒ sequent ⇒ (sequent fset) option› where
  ‹eff Idle (A, B) =
    Some {| (A, B) |}›
| ‹eff (Axiom n ts) (A, B) = (if ‡n ts [∈] A ∧ ‡n ts [∈] B then
    Some {||} else None)›
| ‹eff FlsL (A, B) = (if ⊥ [∈] A then
    Some {||} else None)›
| ‹eff FlsR (A, B) = (if ⊥ [∈] B then
    Some {| (A, B [÷] ⊥) |} else None)›
| ‹eff (ImpL p q) (A, B) = (if (p ⟶ q) [∈] A then
    Some {| (A [÷] (p ⟶ q), p # B), (q # A [÷] (p ⟶ q), B) |} else None)›
| ‹eff (ImpR p q) (A, B) = (if (p ⟶ q) [∈] B then
    Some {| (p # A, q # B [÷] (p ⟶ q)) |} else None)›
| ‹eff (UniL t p) (A, B) = (if ∀p [∈] A then
    Some {| (p⟨t/0⟩ # A, B) |} else None)›
| ‹eff (UniR p) (A, B) = (if ∀p [∈] B then
    Some {| (A, p⟨#(fresh (A @ B))/0⟩ # B [÷] ∀p) |} else None)›
  by pat_completeness auto
termination by (relation ‹measure size›) standard
```

Isabelle/HOL Details III

Our rules don't step on each other (only *r* can *disable r*):

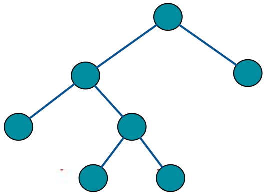
```
lemma per_rules':
  assumes ‹enabled r (A, B)› ‹¬ enabled r (A', B')›
    ‹eff r' (A, B) = Some ss'› ‹(A', B') |∈| ss'›
  shows ‹r' = r›
```

If we give this lemma (+ UNIV_rules) to Blanchette et al., they give us a prover:

```
definition ‹prover ≡ mkTree rules›

codatatype 'a tree = Node (root: 'a) (cont: "'a tree fset")

primcorec mkTree where
  "root (mkTree rs s) = (s, (shd (trim rs s)))"
| "cont (mkTree rs s) = fimage (mkTree (stl (trim rs s))) (pickEff (shd (trim rs s)) s)"
```



Isabelle/HOL Details IV

Blanchette et al. also tell us the prover produces one of two things:

```
lemma epath_prover:
  fixes A B :: ‹fm list›
  defines ‹t ≡ prover (A, B)›
  shows ‹(fst (root t) = (A, B) ∧ wf t ∧ tfinite t) ∨
    (∃steps. fst (shd steps) = (A, B) ∧ epath steps ∧ Saturated steps)› (is ‹?A ∨ ?B›)
```

- A finite, well formed proof tree
  - Soundness: show that this guarantees *validity* of the formula
- a saturated escape path
  - Completeness: show that this induces a *counter model* for the formula

Details omitted here (even though they are interesting!)

References

My prover + formalization:

https://www.isa-afp.org/entries/FOL_Seq_Calc3.html

The abstract completeness framework by Blanchette et al.:

https://www.isa-afp.org/entries/Abstract_Completeness.html